Formal Methods in Computer-Aided Design 2021

Celestial: A Smart Contracts Verification Framework

Samvid Dharanikota*
Microsoft Research India
Bangalore, India
samvid.dharani@gmail.com

Suvam Mukherjee*
Microsoft Corporation
Redmond, USA
sumukherjee@microsoft.com

Chandrika Bhardwaj#
Goldman Sachs
Bangalore, India
chandrika.bhardwaj@gs.com

Aseem Rastogi
Microsoft Research India
Bangalore, India
aseemr@microsoft.com

Akash Lal
Microsoft Research India
Bangalore, India
akashl@microsoft.com

***Abstract*—We present CELESTIAL, a framework for formally verifying smart contracts written in the Solidity language for the Ethereum blockchain. CELESTIAL allows programmers to write expressive functional specifications for their contracts. It translates the contracts and the specifications to F* to formally verify, against an F* model of the blockchain semantics, that the contracts meet their specifications. Once the verification succeeds, CELESTIAL performs an erasure of the specifications to generate Solidity code for execution on the Ethereum blockchain. We use CELESTIAL to verify several real-world smart contracts from different application domains. Our experience shows that CELESTIAL is a valuable tool for writing high-assurance smart contracts.**

***Index Terms*—Smart contracts, Blockchain, Reliability, Testing**

I. INTRODUCTION

Smart contracts are programs that enforce agreements between parties transacting over a blockchain. Till date, more than a million smart contracts have been deployed on the Ethereum blockchain with applications such as digital wallets, tokens, auctions, and games, holding digital assests worth over $200 billion [19].

The most popular language for smart contract development is Solidity [20]. Solidity contracts are compiled to Ethereum Virtual Machine (EVM) bytecode for execution on the blockchain. Unfortunately, Solidity has obscure operational semantics understood only partially by most programmers. This often leaves vulnerabilities in the smart contracts. Repeated high-profile attacks (e.g. `TheDAO` [17] and `ParityWallet` [18] attacks) orchestrated around these vulnerabilities have resulted in financial losses running into millions of dollars. Worse, smart contracts are "burned" into the blockchain on deployment, which does not allow subsequent patches to fix the vulnerabilities. As a result, it is necessary to ensure correctness at the time of deployment.

Smart contracts are relatively small pieces of code with simple data-structures [29]. All these qualities combined—their critical nature, immutability after deployment, and small size—make smart contracts a good fit for formal verification. The challenge, however, is to lower the formal verification entry barrier for smart contracts developers.

Towards that goal, we present CELESTIAL§, an open-source framework for developing formally verified smart contracts. CELESTIAL allows programmers to annotate their Solidity contracts with Hoare-style specifications [32] capturing functional correctness properties. The contracts and the specifications are translated to F* [45], which in an *automated manner*, proves that the contracts meet their specifications. Once F* returns a verified verdict, CELESTIAL erases the specifications from the input contracts, and emits Solidity code that can be deployed and executed on the Ethereum blockchain. By using Solidity as the source language, and providing fully-automated verification, CELESTIAL ensures a low entry barrier for smart contract developers.

F* is a proof assistant and program verifier with a fully dependent type system. We find it suitable for smart contract verification for several reasons. First, it provides SMT-based automation which, as we show empirically, suffices for fully-automated verification of real-world smart contracts. Second, F* supports user-defined effects, allowing us to work in a custom state and exception effect [21] modeling the blockchain semantics. Finally, F* supports expressive higher-order specifications, though we use its first-order subset with quantifiers and arithmetic (adding our own libraries for arrays and maps).

We evaluate CELESTIAL by verifying several real-world Solidity smart contracts that together currently hold millions of dollars of financial assets. The contracts span different application domains including tokens, wallets, and a governance protocol for Azure Blockchain. We studied the contracts (and in some cases, discussed with the developers) to design their specifications and formally verified that the contracts meet those specifications. In the process, we uncovered bugs in some cases (e.g. missing overflow checks), manifesting as F* verification failure. Once we fixed those bugs (e.g. by adding runtime checks), F* was able to successfully verify the contracts in all the cases. The overhead of any additional

*Equal contribution
#Work done during an internship at Microsoft Research India.

§https://github.com/microsoft/verisol/tree/celestial/Celestial

https://doi.org/

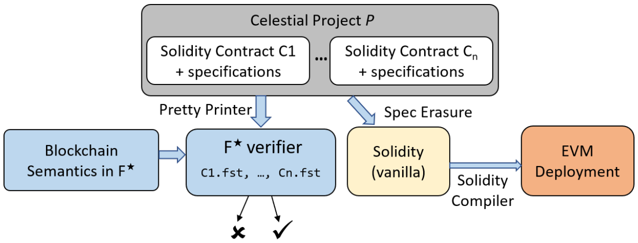

Fig. 1: Architecture of the CELESTIAL framework.

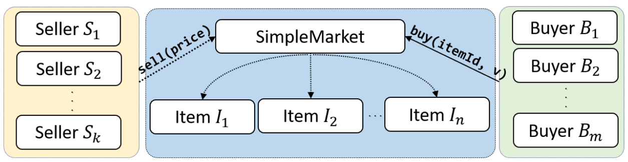

Fig. 2: A simple blockchain based e-commerce application.

instrumentation, which was required for correctness, was at most 20% in terms of gas consumption.

Summarizing our main contributions:

1) We present CELESTIAL, a framework for developing verified Solidity smart contracts. CELESTIAL allows annotatation of Solidity contracts with specifications, and verifies them, in an automated manner, using F*.
2) We evaluate CELESTIAL by verifying functional correctness of several real-world, high-valued smart contracts.

II. OVERVIEW

The high-level architecture of CELESTIAL is outlined in Figure 1. A CELESTIAL project is a set of contracts (e.g. C1, C2, etc. in the figure) written in Solidity. These contracts may be annotated with functional specifications encoding properties of interest. CELESTIAL provides two kinds of translations for these contracts. The first one translates the contracts and their specifications to F* [45], a dependently-typed functional programming language designed for program verification. F*, using a model of the blockchain semantics (Section III), verifies that the contracts meet their specifications. A second translation simply erases all specifications to emit vanilla Solidity contracts. In this section, we use a simple application (Section II-A) to describe the specification language of CELESTIAL (Section II-B). We discuss the verification scope and limitations of the framework later in Section II-C.

A. SIMPLEMART

Consider a simple blockchain-based e-commerce application SIMPLEMART from Figure 2. The application contains a `SimpleMarket` contract (Listing 1) which interacts with one or more buyers and sellers that may either be smart contracts themselves or externally-owned accounts. A seller registers an item for sale by invoking the `sell` method of `SimpleMarket`, with the price as argument. In response, `SimpleMarket` creates an instance of the `Item` contract, which holds metadata about the new item available for sale. It

```
contract SimpleMarket {
  mapping(address => uint) sellerCredits;
  mapping(address => Item) itemsToSell;
  uint totalCredits;
  event eNewItem (address, address);
  event eItemSold (address, address);

  function sell (uint price) public
    returns (address itemId) {
    Item item = new Item(address(this),msg.sender,price);
    itemId = address(item);
    itemsToSell[address(item)] = item;
    emit eNewItem(msg.sender, itemId);
  }
  function buy (address itemId) public payable
    returns (address seller) {
    Item item = itemsToSell[itemId];
    if (item == null) { revert ("No such item"); }
    if (msg.value != item.getPrice())
      { revert ("Incorrect price"); }
    seller = item.getSeller();
    totalCredits = safe_add (totalCredits, msg.value);
    sellerCredits[seller] =
      sellerCredits[seller] + msg.value;
    delete (itemsToSell[itemId]);
    emit eItemSold(msg.sender, itemId);
  }
  function withdraw (uint amount) public {
    if (sellerCredits[msg.sender] >= amount) {
      msg.sender.transfer(amount);
      sellerCredits[msg.sender] -= amount;
      totalCredits -= amount;
    } else { revert ("Insufficient balance"); }
  }
}
```

Listing 1: The `SimpleMarket` Solidity contract

also emits an event (`eNewItem`) informing the seller about the idenity (in this case, the address) of the new item. A buyer may purchase an item by invoking the `buy` method of `SimpleMarket`, passing the item address as an argument, along with the ether amount matching the item price. If the item has not been sold already, `SimpleMarket` records the sale in its state, which involves adding the ether towards the total sales proceeds for the respective seller and marking the item as being sold. The seller may then withdraw the ether from `SimpleMarket` via the `withdraw` method.

Functional correctness of the `buy` method requires that if a buyer initiates `buy` with a valid item and price, then the item is sold and the seller sales proceeds are credited, leaving all other sellers' proceeds unchanged. In addition, we would also like to verify that the call does not result in arithmetic overflow of the seller's proceeds because this can result in honest sellers losing their credits.

B. Specification Language

Listing 2 shows excerpts of the CELESTIAL versions of `Item` and `SimpleMarket` contracts. The general form of a CELESTIAL contract is shown in Listing 3. These annotations are Hoare-style specifications, similar to languages like Dafny [36]. The specifications are written over the contract fields, function arguments, as well as implicit variables such as `balance` (the contract balance), `value` (ether value in a payable method), and `log` (the transaction event log, formally modeled as a list of events). Our specifications cover the full power of first-order reasoning with quantifiers, along with

```
1 contract Item {
2   address seller; uint price; address market;
3   function getSeller () returns (address s)
4     modifies []
5     post (s == seller)
6   { return seller; }
7   // other methods
8 }
9 contract SimpleMarketplace {
10   // contract fields
11   ...
12   invariant balanceAndSellerCredits {
13     balance == totalCredits &&
14     totalCredits >= sum_mapping (sellerCredits)
15   }
16   function buy (address itemId) public
17     returns (address seller)
18     modifies [sellerCredits, totalCredits, itemsToSell,
          log]
19     tx_reverts !(itemId in itemsToSell)
20                || msg.value != itemsToSell[itemId].price
21                || msg.value + totalCredits > uint_max
22     post (!(itemId in itemsToSell)
23          && sellerCredits == old(sellerCredits)[
24            seller => old(sellerCredits)[seller] + msg.
                 value]
25          && log == (eItemSold, msg.sender, itemId)::old(
               log))
26   { // implementation of the buy function }
27 }
```

Listing 2: `Item` and `SimpleMarket` CELESTIAL contracts

```
contract A {
  uint x, y; // fields, as usual

  invariant { ϕ1 } // contract-level invariant

  function foo () public
    modifies [x] // fields that are modified
    tx_reverts ϕ2 // revert condition (under-specified)
    pre ϕ3 // precondition
    post ϕ4 // postcondition
  { s } // Solidity implementation
}
```

Listing 3: A representative CELESTIAL contract

theories for arithmetic (both modular and non-modular), arrays and maps. We provide programmers the ability to write pure functions that can be invoked only from specifications, not Solidity methods, to enable code reuse. We now explain the individual elements of CELESTIAL specifications.

a) Contract invariant: Contract invariant is a predicate on the state of the contract (i.e. its field values) that is expected to be valid at the boundaries of its public methods. When verifying a contract, the invariant is added to the pre- and postconditions of every public method. All contract fields in a CELESTIAL contract are necessarily private (see Section II-C). Additionally, CELESTIAL ensures that all its contracts are *external callback free* (Section IV) to disallow re-entrancy based attacks from external contracts. Hence, it is safe to assume the invariant at the beginning of public methods. Constructors are special; they only guarantee invariant in their postcondition but don't assume it as a precondition. For example, the invariant on line 12 in Listing 2 specifies that the contract's `balance` equals or exceeds the total proceeds from sales which has not been already claimed by the respective sellers (`sum_mapping` is a library function for summing values in an int-valued map).

b) Field updates: The `modifies` clause specifies contract fields that a method can update. The `getSeller` method in `Item` has an empty `modifies` clause (line 4 in Listing 2), which specifies that the function may read the state of the contract, but cannot make any updates.

c) Pre- and postconditions: Preconditions (`pre`) are properties that hold at the beginning of a method execution. Public methods must have a trivial precondition `true` because they can be invoked by the untrusted external world. Postconditions (`post`) are properties that hold when the method terminates successfully (without reverting). The postconditions may refer to field values at the beginning of the method using the `old` keyword. For example, the condition in line 23 in Listing 2 specifies that the final `sellerCredits` is the original `sellerCredits` map with only the `seller` key updated.

d) Revert conditions: `tx_reverts` under-specifies the conditions under which a method reverts, i.e. if `tx_reverts` holds at the beginning of a method, the method will definitely revert. For example, the `buy` function definitely reverts if the buyer invokes it with an item which is not available for sale, or the buyer provides ether which does not match the item price, or the `totalCredits` overflows. This is captured in the specification in line 19. Not specifying `tx_reverts` is equivalent to `tx_reverts(false)`.

e) Safe Arithmetic: In Solidity, arithmetic operations may silently over- or underflow, whereas division by 0 results in reverts. CELESTIAL, when translating to F*, adds assertions before every arithmetic operation which check for no over- and underflows, and division by 0. The programmer must add specifications or runtime checks to allow the verifier to prove the safety of the arithmetic operations. CELESTIAL also provides a safe arithmetic library with built-in runtime checks (`safe_add` operation in line 22 of Listing 1).

To summarize, we have expressed the following properties of the `buy` method. The revert condition specifies that the method reverts when the item is not present or the ether sent by the buyer does not match the item price. The method also reverts when `totalCredits` overflows. Since an invariant of the contract is that `totalCredits` is greater than the sum of pending credits of all the sellers, when `totalCredits` does not overflow, individual seller credits also don't overflow. Finally, line 23 in Listing 2 specifies that only the item seller's credits are incremented by price of the item, while credits for all other sellers remain same.

C. Verification Scope and Limitations

a) Threat model: All contracts and user accounts that are not part of a CELESTIAL project P are treated as the *external world* for P. The external world is free to initiate arbitrary transactions by calling public methods of P with arbitrary arguments. The external world, however, cannot directly access the private fields and methods of P.

b) Trusted Computing Base: The TCB of CELESTIAL includes the CELESTIAL compiler consisting of the two syntax translations, the F* model of the blockchain (Section III), the

F⋆ toolchain itself, and the Solidity compiler (these components are colored blue in Figure 1). With these components in our TCB, formal verification of smart contracts in CELESTIAL guarantees that when the compiled Solidity contracts are run on the blockchain, they behave as per their specifications. We leave it as future work to minimize trust on our F⋆ blockchain semantics (say, by testing it against a Solidity test suite).

c) Solidity Language Restrictions: CELESTIAL does not support `delegatecall` which is used to call functions from other contracts in a way that the callee may directly change the state of the calling address, thereby breaking the function call abstraction. Since this is insecure (for example, the `ParityWallet` [18] attack exploited it), the secure development recommendations suggest against its use [3]. CELESTIAL also does not support embedding EVM assembly. To check the prevalence of these features in real-world contracts, we performed an empirical study. In summary, we found that not more than 45% of highly used and highly valued contracts use these features, and even then in controlled manner where their usage is restricted to a small set of libraries.

d) Modeling Limitations: Our F⋆ semantics does not model gas consumption. As a result, CELESTIAL contracts may revert due to out-of-gas exceptions. The model also does not cover low-level failures such callstack depth overflow. However, these failures can only cause the transaction to revert and therefore do not compromise the verification guarantees. Since we do not model all runtime exceptions, this is one of the reasons that the `tx_reverts` condition for a function is an under-specification for when the function may revert. We also do not precisely model block-level parameters such as timestamp.

III. VERIFYING CELESTIAL CONTRACTS IN F⋆

CELESTIAL compiles the contracts and their specifications to F⋆, which are then verified against a trusted F⋆ library modeling the blockchain semantics. The library consists of the definition of the blockchain state datatype and a custom F⋆ *effect* that encapsulates this state behind the abstraction of an effect layer. We have carefully designed this abstraction to ensure that the verification is scalable and fully automated. The contracts call the stateful API exported by the library and specify precise changes to the blockchain state in their pre- and postconditions, that are verified by F⋆.

A. Blockchain state

We model the blockchain state as consisting of 3 main elements: (a) state of all the contracts (i.e. values of the contract fields), (b) contract balances, and (c) an event log. Since in CELESTIAL all contract fields are private, a contract can directly read or write only its own fields, while interacting with the other contracts through method calls. The event log models the per-transaction event log of the Ethereum blockchain; contracts can use the Solidity `emit` API to output events to this log.

a) Contracts state: We model the state of all the contracts in the blockchain as a heterogeneous map from addresses to records, where the record corresponding to a contract instance contains the values of all its fields. For the `Item` contract from Listing 2, the record type would be:

```
type item_t = { market : address; seller : address; price : uint }
```

Below is the API provided by the contract map (`#` parameters are implicit parameters inferred by F⋆ at the call sites):

```
type address = uint (* 256 bit unsigned integers *)
val contract (a:Type) : Type (* a is the record of contract fields *)
val cmap : Type (* the heterogeneous contracts map *)

val live (#a:Type) (c:contract a) (m:cmap) : prop
val sel (#a:Type) (c:contract a) (m:cmap{live c m}) : a
val create (#a:Type) (m:cmap) (x:a) : contract a & cmap
val upd (#a:Type) (c:contract a) (m:cmap{live c m}) (x:a) : cmap
val addr_of (#a:Type) (c:contract a) : address
```

The API defines the type `address` as 256 bit unsigned integers. The `contract` type is parametric over the record type `a` that contains all the contract fields; for the `Item` contract, type `a` will be instantiated with `item_t`. Type `cmap` is the heterogeneous contracts map type.

The `sel` function returns the `a`-typed record value mapped to a contract instance in the map. The API requires that the contract be `live` in the map (type `m:cmap{live c m}` is a refinement type that requires that the `m` argument at the call sites satisfies `live c m`). The liveness requirement basically says that the contract must be present in the contracts map, preventing `sel` to be called with arbitrary addresses. The `create` function returns the freshly created contract and the new `cmap` that includes a mapping for the new contract, internally assigning a fresh address to the new contract. The API is fully implemented in F⋆, we elide the implementation details for space reasons; all of our development is available online at https://github.com/microsoft/verisol/tree/celestial/Celestial.

b) Contracts balance: We model the contracts balance using a map from addresses to `uint` (the type of 256-bits unsigned integers). An alternative would have been to add `balance` as another one of the contract fields (thus maintaining them as part of the contracts map), but a separate map allows us to specify the balances for external accounts, that do not have an entry in the contracts map.

c) Event log: The event log is a list of events, where each event records the destination address, a string for event type, and a payload (`a:Type & a` is a dependent tuple that packages a `Type` and a value of that type):

```
type event = { to : address; ev_typ : string; payload : (a:Type & a) }
type log = list event
```

With these components, the blockchain state is the following record type:

```
type bstate = { cmap : cmap; balances : Map.t address uint; log : log }
```

B. Libraries for arrays and maps

We have implemented F⋆ libraries for modeling Solidity arrays and maps—the uses of arrays and maps in CELESTIAL contracts are translated to uses of these F⋆ libraries.

Our current implementation only supports dynamically-sized arrays for now, support for compile time fixed-sized arrays is future work. The libraries export operations that match the corresponding Solidity API, and several lemmas that enable the contracts to reason about their properties. For example, following is a snippet of our array library:

```
val array (a:Type) : Type (* an array with element type a *)
val push (#a:Type) (s:array a{length s < uint_max}) (x:a) : array a
val push_length (#a:Type) (s:array a{length s < uint_max}) (x:a)
  : Lemma (requires ⊤) (ensures (length (push s x) == length s + 1))
```

C. An F* effect for contracts

Having set up the model for the blockchain state, we now add a layer on top so that the contracts may manipulate the state and precisely specify the modifications in pre- and post-conditions, while making sure that the verification complexity does not get out-of-hands. We leverage the type-and-effect system of F* for this purpose.

F* distinguishes value types such as uint from *computation types*. Computation types specify the effect of a computation, its result type, and optionally some specifications (e.g. pre- and postconditions) for the computation. For example, Tot uint classifies pure, terminating computations that return a uint value. Similarly uint → Tot uint is the type of pure, terminating functions that take a uint argument and return a uint result. uint → uint is a shorthand for uint → Tot uint; all the blockchain state functions that we have seen so far have an implicit Tot effect.

Following Ahman et al. [21], a state and exception effect for computations that operate on mutable state and may throw exceptions is as follows (st is the type of mutable state):

```
type result (a:Type) = (* the return type of the computations *)
  | Success : x:a → result a
  | Error : e:string → result a

effect STEXN a st (pre:st → prop) (post:st → result a → st →
prop) = ...
```

The semantics of the computations in the STEXN effect may be understood as follows: a computation e of type STEXN a st pre post when run in an initial state (s_0:st) satisfying pre s_0, terminates either by throwing an exception (modeled as returning an Error-valued result) or by returning a value of type a (modeled as returning Success-valued result). In either case, the final state (s_1:st) is such that post s_0 r s_1 holds, where r is the return value of the computation. F* also supports divergent effects, in which case the computations are also allowed to diverge. The STEXN effect in F* comes with a program logic for verifying such computations.

a) Customizing STEXN for contracts: Contract computations naturally fall into the state and exception effect; they read from and write to the mutable blockchain state, and they may throw an exception by calling revert.

However, the revert operation in Ethereum is slightly different from exceptions in, say, OCaml in that it also reverts the underlying state to what it was at the beginning of the transaction, while in OCaml, the state changes are retained. To accommodate this, we instantiate the state st in STEXN with

```
type st = { tx_begin : bstate; current : bstate }
```

where the field tx_begin snapshots the state at the beginning of a transaction. Contracts modify the current state, unless they revert, in which case the current state is reset to tx_begin. Thus, we define the ETH effect for smart contracts as follows:

```
(* state + exception with st as the state *)
effect ETH (a:Type) (pre:st → prop) (post:st → result a → prop) =
  STEXN a st pre post
```

Using ETH effect, we implement the APIs for begin_transaction, revert, and commit_transaction as follows:

```
let begin_transaction () : ETH unit (requires λ_ → ⊤)
(ensures λs0 r s1 → is_success r ∧ s0 == s1) = () (* no op *)

let revert () : ETH unit (requires λ_ → ⊤)
(ensures λs0 r s1 → is_err r ∧ s1=={s0 with current=s0.tx_begin}) = ...

let commit_transaction () : ETH unit (requires λ_ → ⊤)
(ensures λs0 r s1 → is_succ r ∧ s1=={s0 with tx_begin=s0.current}) = ...
```

The function begin_transaction is a no-op, its precondition is trivial (⊤), while its postcondition states that it does not revert (is_success r) and it leaves the state unchanged (s0 == s1). revert, on the other hand, returns an error value, and its output state s1 is same as its input state s0 with current component replaced with the snapshot s0.tx_begin, i.e. the state at the beginning of the transaction. commit_transaction is opposite, it replaces the tx_begin component with s0.current to commit the current state.

The function to get the current state for a contract is as follows, note that the contract is selected from the current component of the state:

```
let get_contract (#a:Type) (c:contract a) : ETH a
  (requires λs → live c s.current.cmap)
  (ensures λs0 x s1 → x == Success (sel c s.current.cmap) ∧
s0 == s1) = ...
```

Similarly, the library provides functions send to transfer balance to a contract and emit to emit an event to the event log.

To make our specifications easier to read and write, we define the following effect abbreviation:

```
effect Eth (a:Type) (pre:bstate → prop) (revert:bstate → prop)
  (post:bstate → a → bstate → prop)
  = ETH a (requires λs → pre s.current)
      (ensures λs0 r s1 →
        (revert s0.current ⟹ Error? r) ∧
        (Success? r ⟹ post s0.current (Success?.x r) s1.current))
```

The pre- and postconditions in the Eth effect are written over the current blockchain state (bstate), as opposed to over the st record. Further, the postcondition is a predicate on a value of type a–it only specifies what happens when the contract function terminates successfully. The revert predicate is a predicate on the input state, which if valid means that the function reverts. We find this abbreviation well-suited for our examples, providing the full-flexibility of the ETH effect to the programmers is of course possible.

CELESTIAL translates each contract to an F* module, where the contract methods are translated to F* functions in the Eth effect. Every function gets explicit parameters for self, sender, value in the case of payable functions, and (underspecified) block-level parameters such as timestamp; after these the function specific parameters follow.

The F⋆ precondition of each function gets to assume the liveness of the contract and the contract invariant. Since these functions can be called by arbitrary, non-verified code, we cannot expect the callers to satisfy more sophisticated preconditions. The postcondition of each function includes the liveness, the contract invariant, and other function-specific postconditions.

The translation of a function body uses the private, per-field getters and setters, also emitted by the translation. Calls to public functions of other contracts are translated to calls to corresponding functions in other F⋆ modules (contracts). Library calls to arrays, maps, etc. translate to corresponding libraries calls in F⋆.

We make a final comment regarding the correctness of the various translations. Since the Celestial source language is just Solidity with specifications, the Celestial to Solidity translation is only spec erasure. The translation to F⋆ is again quite systematic, and therefore, amenable to auditing. Formally proving that the Celestial to F⋆ translation is semantics preserving is an interesting and challenging future work.

IV. Implementing Celestial

The translators to F⋆, for specifications as well as implementation, are combined 2300 lines of Python code. The spec-erasing translator to Solidity is about 750 lines of Python code. The blockchain model is around 1200 lines of F⋆ code. We target the 0.6.8 version of the Solidity compiler for generating EVM bytecode. To aid developer experience, we have written a plugin for Visual Studio Code [16] that supports full syntax highlighting for Celestial. If developers require access to the Celestial specifications in the generated Solidity, we can easily tweak the Celestial to Solidity translation to preserve the specifications as comments.

Limitations: We focused our implementation efforts on Solidity constructs used in our case studies. We currently do not support syntactic features such as inheritance, abstract contracts and tuple types. These mostly only provide syntactic sugar that should be easy to support in future versions of Celestial. Our implementation currently also does not support passing arrays and structs as arguments to functions. While our implementation allows loops in contract functions, we currently do not support writing loop invariants. We also only provide weak specifications for block level constructs (such as `timestamp`, `number` and `gaslimit`), transaction level constructs (such as `origin` and `gasprice`), and functions for obtaining hashes (such as `keccak256` and `sha256`).

Contract Local Reasoning: Calling external contracts can lead to *reentrant* behavior where the external contract calls back into the caller, which is often difficult to reason about. Celestial disallows such behaviors by checking for *external callback freedom* (ECF) [28], [42] which states that every contract execution that contains a reentrant callback is *equivalent* to some behavior with no reentrancy. When this property holds, it is sufficient to reason about non-reentrant behaviors only: any specification over those set of behaviors will hold for all behaviors as well. Thus, ECF allows for contract-local reasoning.

```
1  contract A {
2      bool lock;
3      function foo () public
4          tx_reverts lock
5      {  if(lock) { revert; } ...  }
6
7      function bar (address x) {
8          lock = true;
9          // external call
10         x.call(...);
11         lock = false;
12         ...
13 }   }
```

Listing 4: Ensuring External Callback Freedom

Celestial has two ways of checking for ECF; one of these must hold for each external call. The first is a lightweight syntactic check from VerX [42]. An external call is deemed ECF compliant if it is guaranteed to only be called at the end of a transaction. In other words, for any public method that may transitively invoke an external call, it must ensure that it does not read or write to the blockchain state after the call. External calls that do not fall in this category must satisfy Celestial's second check that asserts that any callbacks made by an external call are guaranteed to revert. We explain this check using the Celestial contract shown in Listing 4. There is an external call in method `bar` on line 10. To prevent reentrancy, the programmer uses a contract field called `lock` and follows the protocol that the `lock` will be assigned true when making an external call. Furthermore, each public method of the contract (such as `foo`) will revert if `lock` is set to *true*. It is easy to see that if the external contracts tries to call back a method of `A`, the transaction will abort.

Celestial's translation to F⋆ adds a sequence of assertions preceding each external call (that does not satisfy Celestial's first check). For each public method of the contract, it takes the `tx_reverts` condition on the method, say ϕ, and inserts `assert` ϕ before the external call. This will ensure that a call back to a public method is guaranteed to revert.

V. Evaluation

We evaluate the development experience with Celestial by writing verified versions of 8 Solidity smart contracts, including real-world contracts spanning crypto-currency tokens, wallets, marketplace, auctions and governance. Some of these contracts are "high-valued", holding millions of dollars of financial assets or having processed millions of transactions.

For each contract, we added detailed functional specifications. If the verification failed, we minimally modified the code in order to discharge the verification conditions. For contracts which required such modifications, we additionally measured the gas consumption overhead, using Truffle [13]. We performed our experiments using an Intel Core i7-7600U dual-core CPU, with 16GB RAM, and running Windows 10. Table I summarizes the various case studies that we performed.

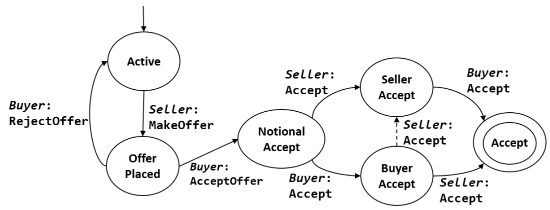

Fig. 3: The AssetTransfer state machine. The dashed arrow indicates a buggy state transition.

Due to lack of space, we discuss details of 3 of the case studies here. We refer interested readers to our Technical Report [25] for a detailed discussion of all the case studies. The sources for all the case studies are available at https://github.com/microsoft/verisol/tree/celestial/Celestial.

			Celestial		
Benchmark	**#C**	**#Sol**	**#Spec**	**#Impl**	**V-Time (sec)**
AssetTransfer*	1	130	70	187	4.26
OpenZeppelin ERC20	4	171	97	200	8.82
BinanceCoin*	2	133	25	136	29.98
WrappedEther*	1	62	62	114	20.00
EtherDelta*	1	281	57	351	63.97
Consensys MultiSig*	2	378	163	289	77.80
SimpleAuction*	1	66	61	101	22.45
Governance Contract	1	417	121	149	86.86

TABLE I: Celestial case studies. We report the number of contracts in the application (#C), LOC of the original Solidity implementation (#Sol), LOC of the Celestial version, divided between specification (#Spec) and implementation (#Impl), and finally the F* verification time (averaged over 3 runs). Benchmarks marked with * used a safe arithmetic library, which is added towards #Impl.

A. AssetTransfer

Application: AssetTransfer [10] is a microbenchmark that provides a smart contract based solution for transferring assets between a buyer and a seller. The contract encodes asset transfer as a finite state machine (FSM) (Figure 3), a common design pattern [11], [39], with the different states denoting the varying stages of approval for the transfer. The contract has notions of *roles*, such as Buyer and Seller, and state transitions are guarded by appropriate roles (for example, the contract can transition from `Active` to `OfferPlaced` when the Seller invokes the `MakeOffer` method).

Specifications. Figure 3 is also the specification for this contract, that is, we must ensure that each of the contract methods respect the transitions mentioned in the FSM diagram. For example, the following is the spec for `MakeOffer`:

```
function MakeOffer (uint _price)
  modifies [sellingPrice, state, log]
  tx_revert (old(state) != Active && msg.sender != Seller)
  post (state == OfferPlaced && sellingPrice == _price)
{ // implementation }
```

The spec ensures that the method makes the correct state transition (`Active` $\rightarrow$ `OfferPlaced`), and this transition can only be caused by the Seller. Interestingly, this spec failed to verify, which led us to discover two bugs in the implementation. These bugs could potentially leave the whole transfer in a frozen state. For instance, one of the bugs led to the erroneous state transition shown in Figure 3. It caused the contract to mistakenly transition to the `SellerAccept` state, even after both the Seller and Buyer had accepted the transfer, which makes the final state (`Accept`) to become unreachable. Fixing these bugs allowed verification to go through. Previous work [47] has noted similar bugs in a different version of the contract. The original contract also had overflow/underflow vulnerabilities, which we eliminated using runtime checks.

Performance. We ran both contracts (Celestial-generated Solidity and original Solidity) through a typical asset-transfer workflow. On an average, the Celestial version consumed $1.12\times$ more gas compared to the original. We account for both the contract as well as any associated library, for instance for safe arithmetic, when measuring the deployment cost.

B. ERC20 Tokens

Application. ERC20 is a standard [4] for Ethereum cryptocurrencies (or *tokens*). Till date, over 400K ERC20 tokens have been deployed on Ethereum, handling financial assets worth *billions of dollars*. We formally verified the OpenZeppelin ERC20 contract [8], which is a popular reference implementation of some of the key ERC20 functions, such as transferring tokens from one account to another and approving third parties to spend tokens on a user's behalf. We also verified the ERC20-based BinanceCoin (BNB) [2] token.

Specifications. We based some of our specifications on earlier efforts to formally verify the OpenZeppelin ERC20 token [6], [47]. The following shows an excerpt. The implementation maintains the balance (number of issued tokens) for each contract address using a `_balances` map. Celestial allows us to easily express the important invariant (line 4) that the sum over the balances for each user equals the total number of tokens issued.

```
1 contract ERC20 {
2   mapping (address => uint) _balances;
3   uint _totalSupply;  // total issued tokens
4   invariant _balanceAndSellerCredits {
5     _totalSupply = sum_mapping(_balances)
6 }}
```

The remaining specifications capture the business logic of key ERC20 functions. The example below shows the postcondition for the `_transfer` method that is used for atomically debiting a source account, and crediting the amount in a destination account. The postcondition ensures that the correct debit and credit operations occur in the source and destination accounts, and all other accounts remain unchanged.

```
function _transfer (address from, address to, uint amt)
  private tx_reverts ..., modifies [...]
  pre _balances[from] >= amt &&
      _balances[to] + amt <= uint_max
  post ite(from == to, _balances == old(_balances),
        _balances == old(_balances)[
           from => old(_balances)[from] - amt,
           to => old(_balances)[to] + amt]))
  { // implementation }
```

The ERC20 token makes copious use of arithmetic operations. OpenZeppelin designed a SafeMath Library [9] to perform runtime checks for overflows and underflows, which the original ERC20 token leverages to ensure runtime safety for arithmetic operations. In contrast, we used the CELESTIAL safe arithmetic operations in public functions, and eliminated runtime checks altogether in private functions when the arithmetic was provably safe.

C. Governance Contract

Application. We study a contract from Microsoft that manages a consortium of mutually-trusted members interacting on a *private* Ethereum blockchain. The contract comprises a set of rules governing operations such as inviting fresh members to join the consortium and adding or removing existing members. The contract is complex, since it maintains many correlated data structures, loops and access control policies, with each logical operation involving intricate changes to multiple data structures. Due to the proprietary nature of the contract, we abstain from showing code or specifications for it explicitly. We did not include several functions in the original contract, whose operations were orthogonal to the governance logic.

Specifications. We briefly describe some of the important properties that we proved.

1) Among members in the consortium, some are designated as being "administrators". An important invariant is that the number of administrators cannot be zero (otherwise the consortium freezes with no further transaction processing).
2) In the contract, logical units of information are maintained in aggregate by several data structures. For example, the contract maintains an array of existing members. However, members can either be referenced by a string identifier, or an address. Thus, the contract maintains a couple of additional mappings that maintain, respectively, associations between string identifiers and addresses, to the correct indices in the array. We specify several invariants to ensure that these data structures are always consistent. For example, we specify that there are no duplicates in the array, no two string identifiers map to the same array index, and the value of each string identifier must not exceed the length of the array of members.
3) We precisely captured the postconditions for operations such as member additions, where we ensure that the operation only updates the necessary keys and indices, while leaving the remaining entries untouched.

We note that some of these properties are similar to those proved by Lahiri et al [35] for a variation of an open-source governance contract [14].

VI. RELATED WORK

The literature on ensuring correctness of smart contracts can be classified into the following broad categories.

Surveys and Best Practices. There is a wealth of available material that highlights known vulnerabilities and exploits in smart contracts [22], [24], [41], [46]. These efforts have resulted in literature suggesting best coding practices for Solidity [5], [12]. CELESTIAL is inspired by these practices, for instance, by ruling out low-level instructions as well as uncontrolled reentrancy, however, the restrictions are not just for avoiding programming pitfalls, but rather to aid semantic verification.

Testing. Frameworks like Truffle [13] allow users to write unit and integration tests for smart contracts in JavaScript. The transactions are typically executed in an in-memory mock of the EVM, such as Ganache [7]. In addition to testing functional behaviors and finding bugs, such tests reveal useful diagnostic information such as gas consumption.

Contract Analysis. A large number of tools have been developed that statically analyze smart contracts (Solidity source code or EVM bytecode) to reveal various vulnerabilities. Examples include MadMax [27] (targeting vulnerabilities due to gas exceptions) and Slither [26] (for identifying security vulnerabilities). Oyente [38] leverages symbolic execution to rule out several classes of vulnerabilities. ContractFuzzer [33] offers a fuzzing based solution for identifying security bugs.

Solythesis [37] is a source-to-source Solidity compiler that instruments the Solidity code with runtime checks to enforce invariants, but specifications particular to each function can't be specified in this framework and it has a significantly high gas overhead because of the runtime checks. VeriSmart [44] offers a highly precise verifier for ensuring arithmetic safety of Ethereum smart contracts, which discovers transaction invariants, but is unable to capture quantified transaction invariants. Tools like teEther [34] leverage symbolic execution to find vulnerable executions and automatically generate exploits.

Each of these tools target a known set of vulnerabilities and offer specialized solutions for them. In contrast, CELESTIAL verifies custom specifications of contracts, relying on verification to rule out all vulnerabilities against that specification.

Formal Verification. VeriSol [35], [47] checks conformance between a state-machine-based workflow and the smart contract implementation, for contracts of Azure Blockchain Workbench [1]. VeriSol does not check for reentrancy; it simply assumes its absence, as opposed to CELESTIAL that enforces it as part of the contract verification. Further, VeriSol does not model arithmetic over/underflow, or check for unsafe type casts, which were an important aspect of our case studies.

VerX [15], [42] is another formal verification tool. VerX uses a syntactic check to ensure ECF (which we use in CELESTIAL as well), however it cannot verify that the program in Listing 4 satisfies ECF. VerX aims for automation of verification by inferring predicates in an abstraction-refinement loop. Such techniques tend to be limited in their ability to reason with quantifiers; VerX uses special built-in predicates like `sum` for quantified reasoning over maps. CELESTIAL, on the other hand, allows for the full power of first-order reasoning with quantifiers. VerX implements its own custom symbolic execution, whereas CELESTIAL uses a simple syntax translation to $F^\star$ and delegates all analysis to the mature $F^\star$ verifier. Unfortunately, the VerX tool is not openly available for further comparisons.

Some verification tools work at the level of EVM bytecode [30], [31], [40], [43], instead of Solidity source level. This is more precise and removes the Solidity compiler from the TCB, however, it is also more time consuming and hard to scale to the larger, complex contracts that we have evaluated in Section V. Bhargavan et al. [23] provide an approach to translate a subset of Solidity to $F^\star$ for verification, as well as a method to decompile EVM bytecode to $F^\star$ to check low-level properties such as establishing worst-case gas bounds for a transaction. Their work is presented as a proof-of-concept only, with limited evaluation and restricted to a small subset of the language.

VII. Conclusion

We presented CELESTIAL, a framework for developing formally verified smart contracts. CELESTIAL provides fully automated verification, using $F^\star$, of Solidity contracts annotated with functional correctness specifications. With the help of several real-world case studies, we conclude that formal verification can be made accessible to smart contract developers for programming high-assurance contracts. Our next steps include enriching our $F^\star$ model of blockchain with more features and validating it using the Solidity testsuite as well as exploring proofs of cross-transaction properties.

References

[1] Azure blockchain workbench. https://azure.microsoft.com/en-us/solutions/blockchain/.

[2] Binance coin. https://www.binance.com/en.

[3] Consensys secure development recommendations. https://consensys.github.io/smart-contract-best-practices/recommendations/.

[4] Eip 20: Erc-20 token standard. https://eips.ethereum.org/EIPS/eip-20.

[5] Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/.

[6] Formal verification of erc20 implementations with verisol. https://forum.openzeppelin.com/t/formal-verification-of-erc20-implementations-with-verisol/1824.

[7] Ganache. https://github.com/trufflesuite/ganache.

[8] Openzeppelin erc20. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol.

[9] Openzeppelin safemath. https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol.

[10] Remix ethereum ide. https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples/asset-transfer.

[11] Solidity docs: State machines. https://solidity.readthedocs.io/en/v0.6.8/common-patterns.html#state-machine.

[12] Solidity security considerations. https://solidity.readthedocs.io/en/v0.6.8/security-considerations.html.

[13] Truffle suite. https://www.trufflesuite.com/.

[14] Validator set contracts. https://github.com/Azure-Samples/blockchain/tree/master/ledger/template/ethereum-on-azure/permissioning-contracts/validation-set.

[15] Verx. https://verx.ch/.

[16] Visual studio code. https://code.visualstudio.com/.

[17] Understanding the dao attack. https://www.coindesk.com/understanding-dao-hack-journalists, 2016.

[18] The parity wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/, 2017.

[19] Etherscan: Contract accounts. https://etherscan.io/accounts/c, 2020.

[20] Solidity v0.7.2. https://solidity.readthedocs.io/en/v0.7.2/, 2020.

[21] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017.

[22] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

[23] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short paper. In Toby C. Murray and Deian Stefan, editors, *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 91–96. ACM, 2016.

[24] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *CoRR*, abs/1908.04507, 2019.

[25] Samvid Dharanikota, Suvam Mukherjee, Chandrika Bhardwaj, Aseem Rastogi, and Akash Lal. Celestial: A smart contracts verification framework. Technical Report MSR-TR-2020-43, Microsoft, December 2020.

[26] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

[27] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.

[28] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[29] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 531–548. ACM, 2019.

[30] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 204–217. IEEE Computer Society, 2018.

[31] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y. A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors, *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, volume 10323 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2017.

[32] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[33] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[34] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1317–1333. USENIX Association, 2018.

[35] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. Formal specification and verification of smart contracts for azure blockchain. *CoRR*, abs/1812.08829, 2018.

[36] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[37] Ao Li, Jemin Andrew Choi, and Fan Long. Securing smart contract with runtime validation. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on*

Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pages 438–453. ACM, 2020.

[38] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[39] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 523–540. Springer, 2018.

[40] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014.

[41] Daniel Pérez and Benjamin Livshits. Smart contract vulnerabilities: Does anyone care? *CoRR*, abs/1902.06710, 2019.

[42] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*, pages 18–20, 2020.

[43] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[44] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1678–1694. IEEE, 2020.

[45] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[46] Antonio Lopez Vivar, Alberto Turégano Castedo, Ana Lucila Sandoval Orozco, and Luis Javier García-Villalba. An analysis of smart contracts security threats alongside existing solutions. *Entropy*, 22(2):203, 2020.

[47] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers*, volume 12031 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2019.